

# Towards JMS Compliant Group Communication

Arnas Kupšys    Stefan Pleisch    André Schiper    Matthias Wiesmann  
École Polytechnique Fédérale de Lausanne (EPFL)  
CH-1015 Lausanne, Switzerland  
Phone: +41-21-693-4240    Fax: +41-21-693-6770  
Email: {firstname.lastname}@epfl.ch

**Abstract**—Group communication provides communication primitives with various semantics and their use greatly simplifies the development of highly available services. However, despite tremendous advances in research and numerous prototypes, group communication stays confined to small niches and academic prototypes. In contrast, message-oriented middleware such as the Java Messaging Service (JMS) is widely used, and has become a de-facto standard. We believe that the lack of standard interfaces is the reason that hinders the deployment of group communication systems.

Since JMS is well-established, an interesting solution is to map group communication primitives onto the JMS API. This requires to adapt the traditional specifications of group communication in order to take into account the features of JMS. The resulting group communication API, together with corresponding specifications, defines group communication primitives compatible with the JMS syntax and semantics.

## I. INTRODUCTION

Group communication has been an active area of research for more than a decade. The notion of process groups, with the possibility to multicast messages to the members of a group, was proposed initially in the context of the V System [1], and later extended by the Isis system to the context of failures [2]. Group communication systems provide *one-to-many* communication primitives with various semantics (e.g., reliable delivery of messages and/or delivery of messages in total order) and their use greatly simplifies the development of highly available services (through replication). Yet, despite tremendous advances in research and numerous prototypes [3], [4], [5], [6], [7], [8], [9], [10], group communication stays confined to small niches and to academic prototypes. Why is this so? Initial group communication systems were monolithic and so were difficult to adapt to specific application needs. However, this argument does not explain the limited use of the technology. Indeed, although recent projects have proposed modular systems, which are more flexible and can be tailored to the application needs [11], [12], [13], this

has not led to significant increase in the use of group communication.

In contrast, there is a communication technology that has recently attracted a lot of interest: the so called message oriented middlewares (MOMs), e.g., MQSeries [14], Tuxedo [15] or JMS (Java Messaging Service) [16]. This technology, which provides abstractions for asynchronous message sending, is increasingly used in industry and is now considered to be an integral part of enterprise computing infrastructure. Some MOMs (e.g., JMS) have become de-facto standards.

The success of MOMs, but also the success of the Web, show that standardized interfaces are a key element for a successful technology. *We believe that the lack of standards is the major reason for the limited use of group communication.* This means that, to become widely used, group communication needs to adapt to the general network environment, and adopt standard interfaces.

What standards do we want for group communication? There is probably no need to invent new standards. As discussed in [17], existing standards can very well be considered for group communication. In this paper we investigate the use of the widely accepted JMS standard for group communication. This study addresses two separate but related issues: (1) the mapping of the group communication API to the JMS interfaces, and (2) the discussion of the semantics of this API in relation with the quality of service that JMS provides. Note that the paper is only about interfaces and specification issues. Implementation of group communication primitives is rather well understood, and is not discussed here.

*Related Work.* Integrating group communication with existing middlewares is not a new idea. For example, group communication has been used for the replication of CORBA objects. Recent examples are the Object Group Service (OGS) [18], Eternal System [19], Interoperable Replication Logic (IRL) [20], Electra [21]. In [22] group communication is used to implement high-available replicated Enterprise Java Beans (EJB) services, and [23] provides causal ordering for JMS

messages.

In contrast, the goal of the paper is different. The paper is about using standard interfaces (namely JMS) to simplify and standardize the usage of group communication (an issue not addressed in the above references).

*Roadmap.* The rest of the paper is structured as follows. Section II gives a brief overview of the JMS notions needed to understand the paper. In Section III, we present the basic idea for the mapping of group communication to the JMS interfaces and discuss how the properties and notions of JMS can be translated to the context of group communication. The core contribution of the paper is in Sections IV and V. In Section IV we first introduce the system model and the definitions, and then give the specification of group communication with respect to JMS. Section V presents the JMS compliant API for group communication. Section VI discusses some additional issues and Section VII concludes the paper.

## II. JAVA MESSAGING SERVICE

The Java Messaging Service (JMS) [16] is a part of Sun Microsystem's Java 2 Enterprise Edition [24]; it is a set of interfaces and associated semantics that govern the access to messaging systems. The basic architecture is shown in Figure 1. JMS assumes a central JMS server, which generally acts as the hub for all communications, and has access to stable storage. The server is transparent to the application, composed of the JMS clients (senders of messages and receivers of messages), and a set of application-defined messages.

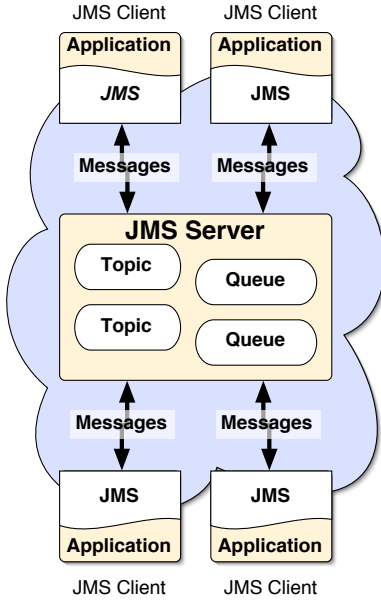


Fig. 1. Basic JMS architecture.

The JMS specification does not define how the server is implemented. It only defines the interfaces and services that the JMS infrastructure must provide.

Two communication paradigms are defined in the JMS specification: *point-to-point* and *publish-subscribe*. In point-to-point messaging, a message is sent by a JMS client to a specified *message queue*, from which it is extracted by another JMS client (which *consumes* or *receives* the message). Hence, the message sent to a message queue is received by only one client. In contrast, publish-subscribe messaging provides one-to-many communication and is based on the concept of *topic*: a message published by a JMS client to a topic is received by all JMS clients that have subscribed to that topic. Note that the publisher does not know the set of subscribers.

Our proposal is to map the group communication API to the JMS publish-subscribe paradigm. Thus in the next paragraph we focus on this paradigm.

### A. JMS Publish-subscribe

JMS specifies two types of subscriptions to the topic: *non-durable* and *durable*. Consider a topic to which a client has subscribed. With a non-durable subscription the client receives messages published to the topic as long as its connection to the server is active. The connection can break (i.e., become inactive) for example because of a link failure, or because of the crash of the client. Messages published after the connection is broken are not guaranteed to be received by the client.<sup>1</sup> In contrast, durable subscriptions mask these failures. Indeed, the client is ensured to receive all messages that have been published to the topic it has subscribed to, even if the connection is not permanently active. Assume, for instance, that the client fails at time  $t_1$  (the failure breaks the connection) and recovers at time  $t_2$ . The JMS server keeps all the messages published when the client connection was “inactive” (time interval  $[t_1, t_2]$ ), and delivers them to the client as soon as its connection is “active” again.

Another JMS feature is the *message delivery mode*, which can be *persistent* or *non-persistent*. Persistent messages are stored by the JMS server on stable storage, and provide guarantees to publishers in case of the crash of the JMS server. If the JMS server receives a persistent messages, it acknowledges the reception to the publisher only after having stored the message on persistent storage. Non-persistent messages, in contrast,

<sup>1</sup>If the connection is broken, the client can try to subscribe again to the topic. Let us assume that the connection was broken at time  $t_1$ , and that a new subscription is received by the JMS server at time  $t_2$ . With non durable subscriptions, the messages published in the interval  $[t_1, t_2]$  are not received by the client.

are not saved on persistent storage, and can thus be lost if the JMS server crashes.

To summarize, as shown in Figure 2, subscription durability specifies a property between (i) the JMS server and (ii) topic subscribers, while persistence is related to the communication between (i) the topic publisher and (ii) the JMS server. Note that durable subscriptions only make sense with persistent messages [16]. In the rest of the paper, we refer to persistence/non-persistence and durability/non-durability as quality of service (QoS).

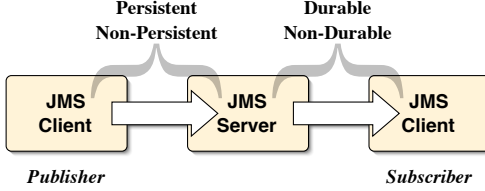


Fig. 2. Persistence vs. durability.

### III. GROUP COMMUNICATION AND JMS: PRELIMINARY CONSIDERATIONS

We now relate group communication and JMS semantics to each other. We start with the mapping of groups onto JMS topics.

#### A. Groups as JMS topics

Our basic idea is to represent *process groups* as *JMS topics*:

- Members of a group  $g$  correspond to the subscribers of the corresponding topic.
- Broadcasting a message to the members of  $g$  corresponds to publishing the message to the corresponding topic.

The idea of representing a group as a topic is quite natural, since JMS uses the notion of a topic to indirectly address a set of JMS clients. Note that representing the group as a JMS queue is less natural, and raises the following semantic issue: while multiple clients can read from the same queue, only *one* client gets a particular message (i.e., if client  $c$  reads message  $m$ , then client  $c'$  cannot read  $m$ ). Queues are therefore not suited to express the multicast semantics of group communication.

#### B. API Mapping

The next question to address is the mapping of group communication primitives onto JMS methods, more specifically onto the methods related to the publish-subscribe paradigm. Clearly, the mapping is not always possible, as some group communication concepts do not exist in JMS.

There are two possible approaches here: (1) rely strictly on the interfaces and standard mechanisms offered by JMS, or (2) add new interfaces to JMS when needed (e.g., for functionality specific to group communication). Both approaches have advantages and drawbacks. Approach (1) has the important advantage not to modify the existing JMS API, whereas approach (2) violates JMS compatibility and thus might confuse developers familiar with JMS. On the other hand, approach (1) might, for some features, not be very natural from the perspective of group communication. Approach (2) does not have this problem.

Consider the following example: in group communication systems, a group member can issue a request to get the current group membership. JMS does not provide an interface for this. So, approach (2) would lead to the addition of a new JMS method to obtain a list of current group members. Approach (1) requires to find another solution.

We have chosen approach (1). By not extending the JMS API for group communication, we believe that we increase the acceptance of our proposal. In Section V, we discuss the work-around that we propose, which is a consequence of our choice to adopt approach (1).

#### C. Open vs. closed groups

In the context of group communication, it is sometimes required that the process that broadcasts a message to a group is part of that group. This is called the *closed* group model. In the *open* group model, no such restriction exists.

In JMS a publisher does not have to be a subscriber to publish to the topic. This corresponds to the open group model. Since the open group model is more general than the closed group model, it seems natural to adopt this model for group communication based on the JMS interfaces.

#### D. Mapping of persistent vs. non-persistent messages

The mapping of group communication primitives to the JMS API is not the only problem that we need to address. We also have to find a mapping for the JMS QoS defined by the persistence/non-persistence of messages and by the durability/non-durability of subscriptions. We start with the persistence issue, and discuss the durability issue in the next section.

Consider a JMS publisher that publishes message  $m$  to topic  $g$ . If  $m$  is persistent, and the publisher received an acknowledgment from the JMS server, then the publisher has the guarantee that message  $m$  will not be lost, even in case of the crash of the JMS server. In contrast, if message  $m$  is non-persistent, then it can be lost if the JMS server crashes. Note that the loss of  $m$  can happen although the publisher does not crash.

If we transpose the second scenario in terms of group communication, we have the following. Consider a process  $p$  that broadcasts message  $m$  to group  $g$ . If the message is not persistent it can be lost, even if  $p$  is correct (i.e., does not crash). The message loss does not happen if the message is persistent. In other words, non-persistent messages provide what is usually called *best-effort* guarantees, while persistent messages can be seen as providing the *strong* guarantees of a reliable (logical) channel between the sender and the group. As group communication traditionally provide more than best-effort guarantees, we assume persistent messages in the rest of the paper.

#### E. Mapping durable vs. non-durable subscription

How does the notion of durable vs. non-durable subscriptions map to guarantees in the context of group communication? This question is more difficult to address than the question of persistence/non-persistence. The reason is that the issue cannot be discussed without referring to what happens to the processes that are members of a group and crash.

In one commonly adopted group communication model, processes that crash are eventually removed from the group. Upon recovery, these processes take a new identity before joining again the group. This model is sometimes called the *crash-no recovery* model: processes that crash seem not to recover, since they recover under a new identity. This model is for example the one of the Isis system [3]. Note that, if a message is broadcast to some group  $g$ , the group communication system has the obligation to deliver messages to members of  $g$ , but only to members of  $g$ . So if a process  $p$  crashes, and is eventually removed from the group, the group communication system stops to have the obligation to deliver messages to  $p$ .

If we transpose this in terms of type of subscriptions, we see that the crash-no recovery model can very nicely be mapped to non-durable subscriptions, in which the JMS server stops to have any obligation toward a subscriber with respect to message delivery if the connection is broken.

If the crash-no recovery model can be mapped to non-durable subscriptions, what is the group communication model that corresponds to durable subscriptions? With durable subscriptions, even if the connection to a subscriber is broken, the JMS server has the obligation to deliver messages to that subscriber. This can be interpreted in the following way in terms of group communication. Let  $p$  be a process member of group  $g$ , and let  $p$  crash at time  $t_1$ , and later recover at time  $t_2$ . Despite of being down during the interval  $[t_1, t_2]$ , process  $p$  delivers all the messages broadcast to the group  $g$ . In other words, although  $p$  crashes, it is not removed from the group.

This means that the group communication system has the obligation to deliver to  $p$  *all* messages broadcast to  $g$ , after  $p$  has became a member of  $g$ . This model is sometimes called the *crash-recovery* model.

To summarize, durable subscriptions can be mapped to a system model in which crashed processes are not removed from the group. Non-durable subscriptions can be mapped to a system model in which crashed processes are eventually removed from the group.

#### F. Clients vs. servers

The JMS architecture distinguishes between the JMS server and JMS clients (see Fig. 1). In the context of group communication, this distinction is rather unusual: for example, the specification of group communication talks only of what JMS calls *clients*. However, the topic of the paper forces us to talk also of the *JMS server*. Even if this is unusual, it has a positive consequence:

- It decouples explicitly the *server(s)* that provide the group communication service from the *clients* that use the service. Note that this decoupling does not prevent a process, in some implementation, to be at the same time a client and a server. This special case is often considered to be the standard case in group communication algorithms. However, an implementation is not forced to adopt this solution. For example, an implementation of group communication could be based on one single (JMS) server. Of course, such an implementation is not fault-tolerant. Another implementation could be based on multiple (JMS) servers, and so be fault-tolerant. Yet, in another implementation, the same process could be both a (JMS) server and a (JMS) client.

The reader should have the decoupling between clients and servers clear in his mind, in order to avoid misunderstanding some issues discussed in the paper. For example, the distinction made above between crash-recovery and crash-no recovery can apply both to (JMS) clients, and to (JMS) servers. However, if one model is chosen for (JMS) clients, this does not impose the same model on (JMS) servers. Moreover, this paper is only about specifications, which means that model issues, discussed in the next section, refer only to (JMS) clients.

### IV. SPECIFICATION OF GROUP COMMUNICATION

The properties ensured by group communication are always defined very rigorously, e.g. [25]. As discussed in the previous section, JMS introduces some new features from the point of view of group communication (for example message persistence, durability of subscriptions), which need to be mapped to the properties of group communication. This has been discussed informally. We explain now how these features can formally be integrated into the specification of group communication.

We first recall some definitions, and then use them in the specification of group communication. The specification of group communication is split into two parts: (1) the specification of the *reliability guarantees* provided by the broadcast primitive, and (2) the additional *ordering guarantees* that can be superimposed on top of the reliability guarantees provided by the broadcast primitive. Since these two issues are orthogonal, we discuss them separately.

#### A. Definitions

1) *Correct and good processes*: In this section, we use the term *process* as synonym for *JMS client*. The guarantees provided by group communication primitives are related to the crash of processes. So, we need some definitions. A process can be *up* or *down*. A process is up if it is operational, and down if it has crashed. A crashed process, after recovery, is again up. However, the specification of group communication is not given in terms of the status up/down of processes at a given time. Instead, the specification refers to the status of process *over their whole execution*. In this context, many specifications of group communication consider that processes do not recover after a crash.<sup>2</sup> In this model, a process that never crashes is said to be *correct* and a process that crashes is said to be *faulty*.

However, because of durable subscriptions, the distinction between correct and faulty processes is not enough. We have to include in our specification the case of processes that crash and later recover. As in [26], we say that a process is *good* if it is eventually always up, i.e., if there is a time  $t$  such that after  $t$  the process is always up.<sup>3</sup> So, a process that crashes only a finite number of times, and recovers after each crash, is a good process. Trivially, a process that never crashes (i.e., is correct) is also a good process. Processes that are not good are said to be *bad*.

2) *Membership views*: A process group corresponds to a JMS topic. Processes can join a group by subscribing to the corresponding JMS topic; they can leave the group by unsubscribing from the corresponding JMS topic. So, the membership of a group changes over time. In group communication, the current group membership is provided to the current group members. The information about the current membership of the group is called the group's *view* (of the membership). So, as processes join or leave some group  $g$ , the membership of  $g$  changes and

the successive views of  $g$  are provided to the processes that are in these views (we say that the views are delivered to the processes). We do not discuss here the precise specification, we only assume that, for every group  $g$ , its successive views are totally ordered: the  $i^{th}$  view of group  $g$  is denoted by  $v_i(g)$ , or simply  $v_i$ . Moreover, we assume that, every process  $p$  delivers the views (to which it belongs) in the index order (e.g., if  $i < j$ , process  $p$  delivers  $v_i$  before  $v_j$ ). For every process, the delivery of each new view is called *view installation*, or *view change*. Note that this specification is called *primary partition membership* [27].

3) *Broadcast vs. partial broadcast*: The specifications of group communication usually consider that the event by which a process broadcasts a message is *atomic*, either fully executed, or not executed at all. This is because it usually does not matter whether the process that executes the broadcast has crashed during the execution of the broadcast primitive, or after. In both cases, because of the crash, there is no obligation for the message to be delivered to the destination processes.

In the context of JMS, the situation is different. This is related to the acknowledgment mechanism provided by JMS (see Sect. III-D). With persistent messages,<sup>4</sup> when some publisher process  $p$  (or JMS client) has received an acknowledgment from the JMS server, we have the guarantee that the message is going to be delivered by the destination processes, *even if  $p$  later crashes*. This leads us to distinguish *broadcast* from *partial broadcast*. Consider some process  $p$  that broadcasts (i.e., publishes) message  $m$ . If  $p$  receives the acknowledgment from the JMS server, we say that  $p$  has *broadcast* message  $m$ . If  $p$  crashes before having received the acknowledgment, we say that  $p$  has *partially broadcast* message  $m$ . Indeed, if no acknowledgment is received by  $p$  before the crash, there is no guarantee that the message is received by the JMS server.

The relation between these two notions and the specifications will become clear in the next paragraph.

#### B. Reliability guarantees of the broadcast primitive

We now formally define the guarantees provided by the broadcast primitive. The properties are expressed in terms of *broadcast* or *partial broadcast*, and *deliver*.<sup>5</sup> Delivery of some message  $m$  is the event by which a message is provided to a process (JMS client). We first discuss the case of non-durable subscriptions, and then

<sup>2</sup>This does not prevent a process from recovering after a crash. However, the consequence is that a process that crashes must recover under a new identity.

<sup>3</sup>It is usual in specification to have properties that are eventually true forever. Actually, from a pragmatic point of view, it is sufficient that the property holds "long enough", where "long enough" depends on the application.

<sup>4</sup>Recall that we have excluded non-persistent messages from our discussion (Sect. III-D).

<sup>5</sup>We could define *partial deliver* as well, but it does not influence the specification.

the case of durable subscriptions.<sup>6</sup> These specifications are adapted from those in [28], which extends the specification in [25] to the case of dynamic groups.

1) *Non-durable subscriptions*: We have explained in Section III-E the link between non-durable subscriptions and the crash/no-recovery model. So, in the case of non-durable subscriptions, the specification distinguishes between correct and faulty processes:

- (P1) *Uniform Validity*: If a process broadcasts message  $m$  to the group  $g$ , then some *correct* process in  $g$  eventually delivers  $m$ , or no process in  $g$  is *correct*.
- (P2) *Uniform Agreement*: If a process  $p$  delivers message  $m$  in view  $v$ , then all processes that are *correct* in  $v$  eventually deliver  $m$ .<sup>7</sup>
- (P3) *Uniform Integrity*: For any message  $m$ , every process in  $g$  delivers  $m$  at most once, and only if  $m$  was previously partially broadcast to  $g$ .
- (P4) *Uniform Same View Delivery*: If two processes  $p$  and  $q$  deliver  $m$ , in view  $v_i$  for  $p$ , and in view  $v_j$  for  $q$ , then  $i = j$ .<sup>8</sup>

The Uniform Validity property (P1) is similar to the one in [25]. It is the property that we need in the open group model (Sect. III-C), i.e., the model in which the process broadcasting a message to group  $g$  does not need to be a member of  $g$ . Note that the property is uniform, which means that the delivery is also ensured if the sender crashes after the broadcast has been executed (see discussion in Sect. IV-A.3).

The Uniform Agreement property (P2) requires agreement on message delivery. While P1 requires that some correct process delivers the message, P2 requires that if some process (correct or not) delivers message  $m$ , then all correct processes also deliver  $m$ .

The Uniform Integrity property (P3) prevents the delivery of duplicate messages. It also requires that the delivery of message  $m$  is justified by a corresponding partial broadcast of  $m$ . Note that a partial broadcast of  $m$  is enough to justify the delivery of  $m$ . If a process broadcasts  $m$ , and crashes during the broadcast, message  $m$  is allowed to be delivered.

The Uniform Same View Delivery property (P4) requires that all processes deliver message  $m$  in the same view. This is a standard property in the context of group communication. The property is sometimes replaced by a stronger property, called *Sending View Delivery* [27].

<sup>6</sup>To simplify the specifications, we assume here that all members of some group  $g$  have the same QoS for the subscription: either all have durable subscriptions, or all have non-durable subscriptions.

<sup>7</sup>The notion of *correct in a view* is explained in [28]. It is out of the scope of this paper to discuss this here.

<sup>8</sup>We say that process  $p$  delivers message  $m$  in view  $v_i$ , if the current view of  $p$  is  $v_i$  when  $m$  is delivered.

However, sending view delivery does not make sense in the open group model.

2) *Durable subscriptions*: In Section III-E we have discussed the link between durable subscriptions and the crash/recovery model. In the case of durable subscriptions, a process  $p$  that crashes at time  $t_1$  and recovers at time  $t_2$ , after recovery is expected to deliver all messages it has missed in the interval  $[t_1, t_2]$ . This requirement can only be expressed if the specification distinguishes between good and bad processes (and not only between correct and faulty processes, as for non-durable subscriptions).

So, for durable subscriptions, we simply replace *correct* by *good* in the properties P1-P4 above (actually only in P1 and P2, since P3 and P4 do not refer to correct processes).

A comment is needed here for the reader familiar with the group communication literature. In most existing group communication systems, if process  $p$  crashes while in some view  $v_i$ , then  $p$  is removed from the group. This means that a new view  $v_{i+1}$  is defined, from which  $p$  is excluded. If  $p$  later recovers, and requests to join again, then a new view  $v_{i+2}$  is defined, which includes  $p$  again. In this case, all messages delivered in view  $v_{i+1}$ , will *not* be delivered by  $p$ . We assume here a different behavior: a process  $p$  that crashes and later recovers, remains a member of the group, even while being down. A process is removed from the group only as a result of an explicit request to leave the group (i.e., unsubscription from the corresponding topic). This is the behavior that users familiar with JMS expect from a durable subscription, and would be surprised not to have similar guarantees in the context of group communication.

### C. Ordering guarantees of the broadcast primitive

After the specification of the reliability guarantees, we specify now additional ordering guarantees for the delivery of messages. Traditionally, the choice is between no ordering requirement (which is called *reliable broadcast*), and total order (called *atomic broadcast*).<sup>9</sup>

There is however a more general and elegant solution; the solution consists in using the group communication primitive called *generic broadcast* [29]. Generic broadcast orders messages according to a *conflict relation*. Generic broadcast ensures that two messages that conflict are delivered in the same order everywhere. Two messages that do not conflict, do not need to be ordered.

Reliable broadcast (no order) and atomic broadcast (total order) are special cases of generic broadcast. Reliable broadcast corresponds to the case where no messages conflict. Atomic broadcast corresponds to the

<sup>9</sup>We do not discuss causal order here.



case where all messages conflict. Moreover, we can define that all messages tagged “reliable broadcast” conflict with all messages tagged “atomic broadcast” (see Table I). This ordering guarantee, which is very useful as illustrated in [30], [29], is not provided by the traditional approach.

TABLE I  
MESSAGE CONFLICT RELATION BETWEEN RELIABLY AND  
ATOMICALLY BROADCAST MESSAGES

		Message $m$	
		Reliable Broadcast	Atomic Broadcast
Message $m'$	Reliable Broadcast	no conflict	conflict
	Atomic Broadcast	conflict	conflict

The ordering guarantee of generic broadcast can be adapted from [28] as follows:

- (P5) *Uniform Generic Order*: If some process delivers message  $m$  in view  $v$  before it delivers message  $m'$ , and the two messages  $m, m'$  conflict, then every process  $p$  that is in view  $v$  delivers  $m'$  only after it has delivered  $m$ .

Note that, the specification (P5) is the same for non-durable and durable subscriptions.

For a process  $p$  that broadcasts a message to the group  $g$ , the “generic broadcast” approach has the following consequence. Instead of choosing a broadcast primitive (reliable broadcast or atomic broadcast), process  $p$  simply tags its message with one of the tags defined for group  $g$  (there can be more than just two tags). The corresponding conflict relation is attached to the group, and defined at group creation time.

## V. MAPPING GROUP COMMUNICATION API TO JMS API

This section describes the JMS compliant API that we propose as an interface for group communication. We map group communication primitives onto JMS methods. As already said in Section III-B, the mapping is not always possible, since some group communication concepts do not exist in JMS. In these cases, we have to find the best work-around.

As mentioned in Section III, there are two possible approaches: (1) rely strictly on the interfaces and standard mechanisms offered by JMS, or (2) add new interfaces to JMS when needed. Since we decided to follow the first approach, we have to find the solutions for problems such as providing views in JMS to group members. Fortunately, JMS provides one extension mechanism:

JMS allows messages to have arbitrary “properties” attached to them. Using this feature, we can for example attach membership information to messages (see below).

Using the same technique, we can map all the group communication primitives to the existing JMS API, and remain fully compliant with the JMS API.

TABLE II  
JOIN AND LEAVE RESTRICTIONS RELATED TO JMS

	Non-durable Subscription	Durable Subscription
$p$ can request $join\ q$	no	no
$p$ can request $leave\ q$	no	yes

Nevertheless, there is one problem that cannot be solved using message properties. The problem is related to the requests to join and to leave a group. Join is mapped to the method to subscribe to a topic, and leave to the method to unsubscribe to a topic. The JMS API does not allow a client  $p$  to request a subscription for another client  $q$ . In group communication systems, a process  $p$  can usually issue a request to add another process  $q$  to the group. The same problem arises for the leave primitive, in the case of non-durable subscriptions. A JMS client  $p$  cannot close the non-durable subscription of another client  $q$  (this is possible for durable subscriptions). In group communication systems, a process  $p$  can usually issue a request to remove another process  $q$  from the group. So, we have to restrict our join and leave group communication primitives to match the JMS interface. The restrictions are summarized in Table II.

### A. JMS Classes

We represent groups as JMS topics and group members as the subscribers to these topics. So, in terms of the JMS API, on the client side a group has an associated TopicSession instance, and each member of the group has an instance of the class TopicSubscriber (Fig. 3). The class TopicPublisher is used to broadcast messages. Remember that the JMS model implies an *open group* model (see Sect. III-C): senders do not have to be part of the group to broadcast messages to it. Message reception can be done either (1) by calling a method of the class TopicSubscriber (the call can be blocking if no message is available, or can return immediately), or (2) by registering a callback. The callback is provided by the interface MessageListener (Fig. 3).

### B. JMS Methods

We divide the JMS methods into two basic categories: *administrative methods* and *communication methods*. Administrative methods are used to set up groups, and

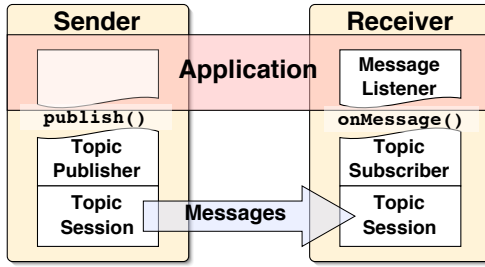


Fig. 3. JMS Classes.

are in general used during the setup phase of the program. Communication methods represent the interface, used for actual communication, i.e., broadcasting and delivering messages. Administrative methods and communication methods can further be characterized as *down* calls or *up* calls. Down calls correspond to usual method calls, and up calls correspond to callbacks. Table III summarizes the API mapping, which is now discussed in more details.

#### 1) Communication methods:

a) *broadcast(g,m)*: The *broadcast* primitive sends a message to all members of a group. In order to broadcast a message  $m$  to some group  $g$ , a client simply calls the method `publish(m)` on the instance of the `TopicPublisher` class that corresponds to  $g$ . The client uses the same interface to send messages, regardless of the type of ordering properties he expects (order or no order). The ordering constraints are defined by the message conflict relation (see Sect. IV-C), and the client just needs to attach the appropriate tag to each message.

b) *deliver(g,m)* — *down call*: In order to deliver a message broadcast to group  $g$ , a client simply calls the method `receive()` on the the instance of the `TopicSubscriber` class that corresponds to  $g$ . The call is blocking if no message is available. Note that another non blocking method, called `receiveNoWait()`, is also available.

c) *deliver(g,m)* — *up call*: In order to deliver a message broadcast to group  $g$ , a client can also register a callback. A callback is provided by the interface `MessageListener`. When a message  $m$  is available for delivery, the method `onMessage(m)` is automatically called.

d) *viewChange(g,m)*: Traditionally group communication systems have a special call to notify of a view change. However, JMS has no such interface. On the other hand, JMS specifies message *headers* and, as mentioned earlier, allows the attachment of *properties* to messages. So, a simple solution is to consider that delivering a new view  $v$  for group  $g$  is like delivering a message  $m$  for group  $g$ . A “view” message is distinguished from a “normal” message by its “JMSType”

header. A “view” message has the header “JMSType” set to the value “new-view”, and has a property called “JMS.view” with a value equal to the new view.

Like for normal messages, a view change message can be received either by a down call, or through an up call (callback).

#### 2) Administrative methods:

a) *createGroup(g)*: Creating a new group corresponds to creating a new JMS topic. Topic creation is outside of the scope of the JMS specification. Each implementation will provide its own mechanism for creating topics (groups).

b) *setMessageConflictRelation(g,conflict)*: As for the creation of groups, the specification of the message conflict relation for some group  $g$  must be handled outside of the JMS API. This is done at group creation time.

c) *joinGroup(g)* — *non-durable subscription*: As explained before, we have to restrict our group communication primitive for joining a group: a process can only add himself to the group. For non-durable subscriptions, the client calls the method `TopicSession.createSubscriber(g)`, where  $g$  is the topic.

d) *joinGroup(g,processName)* — *durable subscription*: Joining a group with durable subscription requires an additional parameter, namely the *processName*. In JMS, this parameter is used to uniquely identify a durable subscription, and must be unique per JMS server. So, to join a group with a durable subscription, the client calls the method `TopicSession.createDurableSubscriber(g,processName)`, where  $g$  is the topic.

e) *leaveGroup(g)* — *non-durable subscription*: We also have to restrict the group communication primitive for leaving a group: a process can only remove himself from the group. For non-durable subscriptions, the client calls the method `close()` on the instance of the `TopicSubscriber` class that corresponds to  $g$ .

f) *leaveGroup(g,processName)* — *durable subscription*: For the durable subscriptions, JMS allows a client to unsubscribe another client (see Table III). To remove a client from the group, the client calls the method `TopicSession.unsubscribe(processName)`. Note that `TopicSession` is not necessary associated with some topic, which implies that *processName* must be unique not only in the group, but in the whole system.

g) *getGroupView(m)*: Traditionally, group communication systems have a call to get the current membership (i.e., view) of the group. JMS does not have such an interface. As already said in the context of the *viewChange* method, messages in JMS can have various “properties”. Like for “view” messages,



TABLE III  
GROUP COMMUNICATION INTERFACE AND JMS METHODS

Primitive	JMS method	Direction	Note
<b>Communication methods</b>			
<code>broadcast(<math>g, m</math>)</code>	<code>TopicPublisher.publish(m)</code>	down	broadcasts a message
<code>deliver(<math>g, m</math>)</code>	<code>m = TopicSubscriber.receive()</code>	down	delivers a message
<code>deliver(<math>g, m</math>)</code>	<code>MessageListener.onMessage(m)</code>	up	delivers a message
<code>viewChange(<math>g, m</math>)</code>	<code>m = TopicSubscriber.receive()</code> with message header <code>JMSType="new-view"</code>	down	notification of view change
<code>viewChange(<math>g, m</math>)</code>	<code>MessageListener.onMessage(m)</code> with message header <code>JMSType="new-view"</code>	up	notification of view change
<b>Administrative methods</b>			
<code>createGroup(<math>g</math>)</code>	outside of the scope of JMS API		creation of a new group
<code>setMessageConflictRelation(<math>g, conflict</math>)</code>	outside of the scope of JMS API		definition of the message conflict relation for group $g$
<code>joinGroup(<math>g</math>)</code>	<code>TopicSession.createSubscriber(g)</code>	down	add myself to the group (non-durable subscription)
<code>joinGroup(<math>g, processName</math>)</code>	<code>TopicSession.createDurableSubscriber(g, processName)</code>	down	add myself to the group (durable subscription)
<code>leaveGroup(<math>g</math>)</code>	<code>TopicSubscriber.close()</code>	down	remove myself from the group (non-durable subscription)
<code>leaveGroup(<math>g, processName</math>)</code>	<code>TopicSession.unsubscribe(processName)</code>	down	remove a process from the group (durable subscription)
<code>getGroupView(<math>m</math>)</code>	<code>m.getStringProperty("JMS.view")</code>	down	Returns the view in which message $m$ was delivered.

we propose to attach to ordinary messages the property “JMS\_View”, whose value is the view in which the message was delivered. So, calling the method `m.getStringProperty(“JMS_View”)` returns the view in which message  $m$  was delivered. To get the current view, the client must call this method on the last message delivered, where the last message is either an “normal” message, or a “view” message.

## VI. RELATED ISSUES

In this section, we discuss some additional issues related to JMS.

### A. Message Priorities

In Section IV, we have defined the ordering property P5. The JMS specification defines an additional mechanism that may affect message ordering, namely message priorities. JMS allows the client to associate priorities to the messages it sends. The JMS specification does not require strict enforcement of guarantees with respect to priorities (it says that an implementation should do its best to respect message priorities). So, priorities can be completely ignored. However, for some applications, priorities can be useful.

Note that the priority mechanism is orthogonal to the order property P5. Message delivery can be ordered according to priorities, as long as this does not lead to the violation of property P5.

### B. Subscription Notifications

The JMS specification defines another mechanism called *subscription notification*. This mechanism allows a publisher (once it has registered to this notification service) to be notified when there are no subscribers, and when there are subscribers again.

The mechanism cannot be used for group communication, to provide the group membership information (e.g., the views). This is because, the mechanism provides information to publishers, whereas, in group communication, the view change information must be provided to the group members (subscribers).

## VII. CONCLUSION

In this paper we have discussed the mapping of the features provided by group communication onto the standard JMS interface. We propose a JMS compliant API for group communication, as well as a specification for group communication that takes into account the quality of service defined by JMS (message persistence and durability of subscriptions).

As the interface looks familiar to JMS developers, we hope that our proposal will contribute to a wider use of the group communication abstractions, and that group communication will become an integral part of future applications. In order to validate our API and specifications, we have started to build a prototype.

**Acknowledgments.** We would like to thank Sam Toueg for discussions related to the specification of group communication.

## REFERENCES

- [1] D. R. Cheriton and W. Zwaenepoel, "Distributed process groups in the V kernel," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 2, pp. 77–107, May 1985.
- [2] K. P. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *Proc. of 11th ACM Symposium on Operating Systems Principles*, 1987, pp. 123–138.
- [3] K. P. Birman, "The process group approach to reliable distributed computing," *Communications of the ACM*, vol. 36, no. 12, pp. 37–53, 1993.
- [4] G. D. Parrington, S. K. Shrivastava, S. M. Wheeler, and M. C. Little, "The design and implementation of Arjuna," Tech. Rep. TR94-65, ESPRIT Basic Research Project BROADCAST, 1994.
- [5] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm, "Phoenix: A toolkit for building fault-tolerant distributed applications in large scale," in *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, Texas, USA, 1995, IEEE, Workshop held during the 7<sup>th</sup> Symposium on Parallel and Distributed Processing, (SPDP-7).
- [6] R. van Renesse, K. P. Birman, and S. Maffei, "Horus: A flexible group communication system," *Communications of the ACM*, vol. 39, no. 4, pp. 76–83, 1996.
- [7] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos, "Totem: a fault-tolerant multicast group communication system," *Communications of the ACM*, vol. 39, no. 4, pp. 54–63, 1996.
- [8] D. Dolev and D. Malki, "The Transis approach to high availability cluster communication," *Communications of the ACM*, vol. 39, no. 4, pp. 64–70, 1996.
- [9] A. Baratloo, P. E. Chung, Y. H. Huang, S. Rangarajan, and S. Yajnik, "Filterfresh: Hot replication of java RMI server objects," in *Proceedings of the 4<sup>th</sup> Conference on Object Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, USA, 1998, USENIX, pp. 59–63.
- [10] K. P. Birman, R. Constable, M. Hayden, C. Kreitz, O. Rodeh, R. van Renesse, and W. Vogels, "The Horus and Ensemble projects: Accomplishments and limitations," in *Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX '00)*, Hilton Head, South Carolina USA, 2000.
- [11] Mark Hayden, "The Ensemble system," Technical Report TR98-1662, Department of Computer Science, Cornell University, Jan. 8, 1998.
- [12] H. Miranda, A. Pinto, and L. Rodrigues, "Appia: A flexible protocol kernel supporting multiple coordinated channels," in *Proceedings of the 21<sup>st</sup> International Conference on Distributed Computing Systems (ICDCS-01)*, Phoenix, Arizona, USA, 2001, pp. 707–710, IEEE Computer Society.
- [13] M. A. Hiltunen and R. D. Schlichting, "The Cactus approach to building configurable middleware services," in *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Nürnberg, Germany, 2000.
- [14] IBM Corp., *MQSeries Application Programming Guide*, New Orchard Road, Armonk, NY 10504 USA, 11 edition, 2000, SC33-0807-10.
- [15] "BEA Tuxedo: The programming model," white paper, BEA Systems, 315 North First Street, San Jose, CA 95131 USA, Nov. 1996.
- [16] M. Hapner, R. Sharma, J. Fialli, and K. Stout, *JMS specification*, Sun Microsystems Inc., 4150 Network Circle, Santa Clara, CA 95054 USA, 1.1 edition, April 2002, <http://java.sun.com/products/jms/docs.html>.
- [17] M. Wiesmann, X. Défago, and A. Schiper, "Group communication based on standard interfaces," in *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA-03)*, Cambridge, MA, USA, 2003, pp. 140–147.
- [18] P. Felber, *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*, Ph.D. thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998.
- [19] P. Narasimhan, *Transparent Fault Tolerance for CORBA*, Ph.D. thesis, University of California, Santa Barbara, USA, September 1999.
- [20] R. Baldoni, C. Marchetti, and A. Termini, "Active Software Replication through a Three-tier Approach," in *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS'02)*, Osaka, Japan, October 13–16 2002, pp. 109–118, IEEE.
- [21] S. Maffei, "Adding group communication and fault-tolerance to corba," in *USENIX Conference on Object-Oriented Technologies*, 1995.
- [22] M. Pasin, M. Riveill, and T. S. Weber, "High-available enterprise JavaBeans using group communication system support," in *Proceedings of the European Research Seminar on Advances in Distributed Systems (ERSADS2001)*, Bologna, Italy, 2001.
- [23] P. Laumay, E. Bruneton, N. de Palma, and S. Krakowiak, "Preserving causality in a scalable message-oriented middleware," in *Proceedings of the Middleware 2001 : IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, November 2001, vol. 2218, pp. 311–329, Lecture Notes in Computer Science, Springer Verlag.
- [24] B. Shannon, *Java 2 Enterprise Edition specification*, Sun Microsystems Inc., 4150 Network Circle, Santa Clara, CA 95054 USA, 1.4 edition, April 2003.
- [25] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Tech. Rep. TR94-1425, CS, University of Toronto; CS, Cornell University, May 1994.
- [26] M. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," *Distributed Computing*, vol. 13, pp. 99–125, 2000.
- [27] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," *ACM Computing Surveys*, vol. 4, no. 33, pp. 1–43, December 2001.
- [28] A. Schiper, "Dynamic Group Communication," Tech. Rep. ID:200327, École Polytechnique Fédérale de Lausanne (EPFL), 2003.
- [29] F. Pedone and A. Schiper, "Handling message semantics with generic broadcast protocols," *Distributed Computing*, vol. 15, no. 2, pp. 97–107, 2002.
- [30] S. Mena, A. Schiper, and P. Wojciechowski, "A Step Towards a New Generation of Group Communication Systems," in *Proceedings of the Int. ACM/IFIP/USENIX Middleware Conference*, Rio de Janeiro, Brazil, June 2003, LNCS 2672, Springer-Verlag.